AutoJudge: Predicting Programming Problem Difficulty

By- Riddhi Sidana | 23322023 |BS-MS Economics

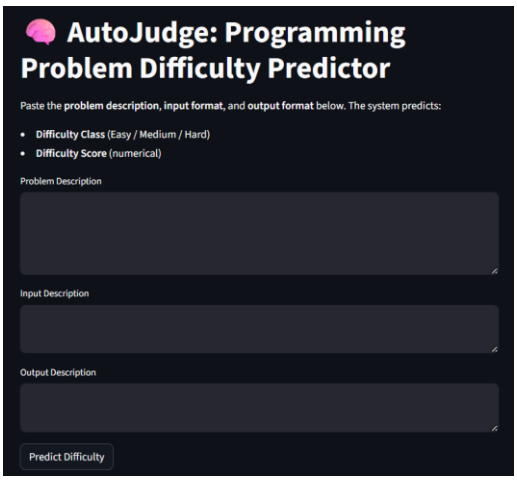
Introduction:

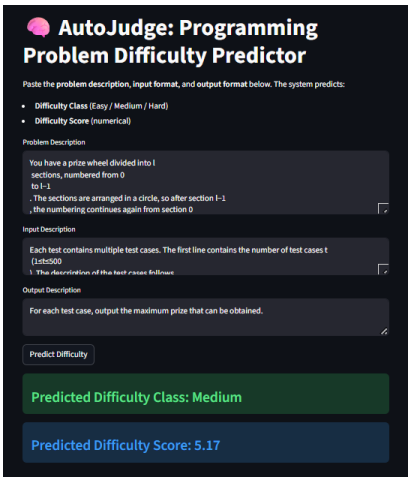
Online competitive programming platforms such as Codeforces, CodeChef, and Kattis categorize programming problems into difficulty levels (Easy, Medium, Hard) and assign numerical difficulty scores. These classifications are largely subjective and depend on human judgment and user feedback, which can vary widely across users.

This project formulates the difficulty prediction task as a **multi-task learning problem**, consisting of:

1. **Multi-class classification** to predict the difficulty category (Easy / Medium / Hard)
2. **Regression** to predict a continuous numerical difficulty score

The system uses **only textual information** from programming problems, such as problem descriptions and input–output specifications, without relying on user interaction data or historical submission statistics.





Dataset Description:

The dataset consists of **4,112 programming problems**, collected via web scraping from competitive programming platforms. Each data sample contains:

- title
- description
- input_description
- output_description
- problem_class (Easy / Medium / Hard)
- problem_score (numerical difficulty score on a scale of 1-10)
- sample_io (sample input and output test cases)
- url (link to the problem statement)

Data Shape: *(4112,8)****, Null Values****: None*

dtypes: *float64 (Problem Score), object (rest)*

Data Preprocessing:

All textual fields (title, description, input_description, output_description) were concatenated to form a single text corpus for each problem.

Text preprocessing included:

- Lowercasing all text
- Removing HTML tags
- Removing special characters and punctuation
- Normalizing whitespace
- Handling missing values using empty strings

✅ This ensured consistency and reduced noise before feature extraction.

Feature Engineering:

1. **TF-IDF Text Features:** Text was converted into numerical features using **TF-IDF Vectorization** with the following configuration:

- Maximum features: 30000
- N-grams: (1, 2)
- Stop words: English
- Sublinear term frequency scaling

✅ These settings capture both individual keywords and meaningful multi-word phrases common in programming problems.

2. **Handcrafted Numeric Features:** In addition to TF-IDF features, domain-specific numeric features were engineered to capture problem complexity more explicitly. These included:

- Text length (log-scaled)
- Count of mathematical symbols
- Constraint awareness flags
    - Presence of constraints
    - Presence of large input sizes (e.g., 10^5, 10^6)
    - Presence of time limits
- **Algorithmic keyword groups**, capturing mentions of:
    - Dynamic Programming, Graph algorithms, Data Structures, String algorithms, Greedy techniques
    - Mathematics, Geometry

All numeric features were scaled using **StandardScaler** and combined with TF-IDF features using sparse matrix concatenation.

Modelling and Evaluation:

All models were trained using a combined feature space consisting of TF-IDF text representations and handcrafted numeric features. Stratified train–test splits were used to preserve class distribution.

Classification (Difficulty Class):

Logistic Regression (L2 regularization), Linear Support Vector Machine (SVM) and Multinomial Naive Bayes (baseline) were evaluated. Hyperparameters were tuned using **GridSearchCV** with **stratified 5-fold cross-validation**, primarily focusing on the regularisation parameter C and class weighting strategies.

Model	Accuracy	Macro F1	Weighted F1
Tuned Logistic Regression	0.52	0.46	0.49
Tuned Linear SVM	**0.54**	**0.50**	**0.51**

Final model chosen: Tuned Linear SVM

This model provided the best balance between accuracy, macro-F1 score, and recall for hard problems.

Classification Performance:

- Accuracy: 0.54
- Macro F1-score: 0.50
- **Easy:** 0.58 recall, indicating good detection of simple problems.
- **Medium:** Lowest recall, reflecting ambiguity between Easy and Hard boundaries.
- **Hard:** Strong recall (0.73), demonstrating the model's ability to identify complex problems.

The results reflect the inherent ambiguity in difficulty labelling, particularly between Easy and Medium problems.

Binary Difficulty Analysis (Hard vs Not-Hard)

To further analyze model behavior, the multi-class problem was reformulated as a binary classification task:

- Hard
- Not-Hard (Easy + Medium)

This analysis is especially relevant for platforms that prioritize filtering or ranking difficult problems.

Binary Classification Results

Model	Accuracy	Hard Recall	Macro F1
Tuned Logistic Regression	0.61	**0.80**	0.60
Tuned Linear SVM	**0.64**	0.73	**0.64**

Interpretation:

- Logistic Regression achieves very high recall for Hard problems but at the expense of increased false positives.
- Linear SVM provides a better precision–recall balance, resulting in higher overall accuracy and macro-F1.

✅ This confirms that Linear SVM is more stable and generalizable for downstream use.

Regression (Difficulty Score):

In addition to categorical prediction, difficulty was modelled as a continuous score to provide a more fine-grained estimate.

For numerical difficulty prediction, the following regression models were evaluated:

- Linear Regression (baseline)
- Random Forest Regressor
- Gradient Boosting Regressor

Hyperparameter tuning was performed using **RandomizedSearchCV** to efficiently explore the parameter space while controlling computational cost.

Regression Results:

Model	MAE	RMSE
Linear Regression	1.997	2.484
Gradient Boosting	1.645	1.954
Random Forest	**1.635**	**1.949**

☑ Random Forest achieves the **lowest MAE and RMSE**, indicating better robustness to noise and feature interactions and, on a difficulty scale of approximately 1–10, an RMSE of ~1.95 suggests the model predicts within ±2 difficulty points on average.

Final model chosen: Random Forest Regressor

Key Insights:

✅ Hard problems show stronger lexical and structural signals, enabling higher recall.

✅ Regression provides a more stable and informative difficulty estimate than discrete classification alone.

✅ Classical machine learning models, when combined with domain-specific feature engineering, can achieve strong performance without deep learning.

✅ Difficulty classification exhibits a performance ceiling due to **label subjectivity**, especially between Easy and Medium problems.

✅ Ensemble tree-based methods significantly outperformed linear baselines, confirming the presence of non-linear relationships in the data.

Web Application:

To demonstrate real-time usability of the proposed system, a **local web application** was developed using **Streamlit**. The application enables users to input a new programming problem description and instantly obtain both categorical and numerical difficulty predictions.

Application Architecture: The web interface loads pre-trained models and preprocessing components, ensuring that:

- No retraining occurs at inference time
- Predictions are fast and reproducible
- The deployment pipeline exactly mirrors the training pipeline

The following components are loaded at runtime:

- TF-IDF Vectorizer (tfidf.pkl)
- StandardScaler for numeric features (scaler.pkl)
- Final classification model (final_classifier.pkl)
- Final regression model (final_regressor.pkl)

All files are stored locally and loaded using **joblib.**

Input Interface: The application provides three text input fields:

- **Problem Description**
- **Input Description**
- **Output Description**

✅ These fields capture the complete textual specification of a programming problem. The inputs are concatenated internally and passed through the same preprocessing and feature extraction pipeline used during model training.

Feature Construction at Inference: Upon user submission:

1. Text is lowercased and normalized.
2. **TF-IDF features** are generated using the pre-fitted vectorizer.
3. **Handcrafted numeric features** are extracted, including:
    - Text length
    - Mathematical symbol density
    - Constraint awareness indicators
    - Algorithm-specific keyword group counts (DP, graph, data structures, etc.)
4. Numeric features are scaled using the saved StandardScaler.
5. Sparse TF-IDF features and scaled numeric features are concatenated to form the final feature vector.

✅ This ensures full consistency between training and deployment.

Model Inference:

The final feature vector is passed to:

- A **Linear SVM classifier** to predict the difficulty class (Easy / Medium / Hard)
- A **Random Forest regressor** to predict a numerical difficulty score

✅ Predictions are performed in real time and returned immediately to the user.

Output Display:

The application displays:

- **Predicted Difficulty Class** (Easy / Medium / Hard)
- **Predicted Difficulty Score** (numerical value, formatted to two decimal places)

✅ Clear visual cues are used to distinguish categorical and numerical outputs, improving interpretability for non-technical users.

Execution and Deployment:

The application is designed for **local execution only**, in line with project requirements. It can be launched using:

streamlit run app.py

✅ The app runs in a browser locally and requires no external hosting, database, or authentication.

Summary: The Streamlit-based web application provides a complete end-to-end demonstration of the AutoJudge system. By integrating trained machine learning models with an intuitive user interface, the application enables practical evaluation of automated problem difficulty prediction using only textual problem descriptions.

Limitations and Future Work:

Limitations:

- Difficulty labels are subjective and platform-dependent
- Overlap between Easy and Medium classes limits classification accuracy
- No user interaction or runtime statistics were available

Future Improvements:

- Incorporating code submissions or solution metadata
- Using transformer-based language models
- Learning platform-specific difficulty scales

Conclusion: AutoJudge successfully demonstrates that programming problem difficulty can be reasonably estimated using textual descriptions alone. While classification accuracy is constrained by label ambiguity, regression models provide stable difficulty estimates.